SkillZip: Evaluation-Free Skill Compression for Self-Evolving Agents by Discovering Reusable Structure

Xiaofan Bai[1,*], Hongqiang Lin[2,*], Chao Liu[1], Yantao Zhang[3],
Xuan Jin[1,†], Xipeng Cao[1], and Yuhong Li[1]

[1]Alibaba Group [2]Zhejiang University [3]Duke University

baixiaofan.bxf@alibaba-inc.com, linhongqiang@zju.edu.cn, chaoheng.lc@alibaba-inc.com, russo.zhang@duke.edu
jinxuan.jx@alibaba-inc.com, caoxipeng.cxp@alibaba-inc.com, daniel.lyh@alibaba-inc.com

*Equal contribution. †Project leader.

Abstract— Self-evolving agents accumulate reusable skills by appending successful procedures and failure fixes. Over time, the same requirement is often restated in several branches, examples, and warnings, while common action sequences are copied rather than reused. The resulting skill becomes expensive to inject and difficult to maintain. Generic prompt compression is ill-suited to this setting because a skill is not a flat passage: its name and description define when it applies, its workflow controls execution, its tool and output contracts constrain validity, and rare exceptions may remain essential even when no sampled task activates them. Evaluation-guided compression can test these behaviors, but it introduces rollouts, cost, and dependence on the compression-time evaluation set.

We present SkillZip, an evaluation-free method that compresses a skill by finding its *shortest faithful structural explanation*. The intuition is "explain once, reference many": state a repeated rule once at the scope where it applies, factor a repeated action sequence into a shared procedure, and keep only the differences as explicit exceptions. We formalize this intuition as a typed minimum-description-length objective over a skill contract and a residual, subject to a hard coverage constraint for every extracted trigger, workflow edge, tool requirement, obligation, and output field. The formulation provides simple sharing thresholds, preserves unique rare rules by construction, and supports efficient local updates. SkillZip has a one-shot mode with one structured extraction call and deterministic optimization, and a continual *Zip-on-Write* mode that integrates each self-evolution patch without replaying tasks or reparsing the full history. Through comprehensive experimental evaluations, we demonstrate the effectiveness and superiority of SkillZip in compression performance, generalizability, and cost overhead.

I. Introduction

A self-evolving agent improves by turning experience into reusable instructions. When a tool fails, it appends a warning; when an answer violates a format, it adds an example; when a rare branch succeeds, it records the successful procedure. Each update is locally reasonable. The accumulated skill, however, is usually edited as an append-only notebook rather than maintained as a coherent program. After enough rounds, "never overwrite the source file" may appear in the introduction, three workflow branches, and an example, while the same validate–repair–verify sequence is copied repeatedly with only minor differences.

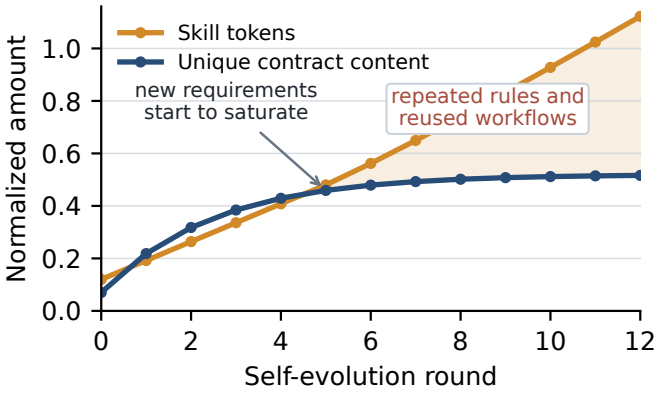

Fig. 1. **The skill growth tendency with respect to self-evolving progress on diverse benchmarks for a CodeX Agent.** Self-evolution can keep increasing skill length after genuinely new procedural content has largely stabilized.

This creates a systematic mismatch between *textual growth* and *procedural growth*. New text keeps accumulating even after the number of genuinely new requirements begins to saturate. Because a loaded skill occupies the context on every invocation, redundant text increases prefill cost and can obscure the instructions that actually govern execution. Figure 1 illustrates the phenomenon that our longitudinal study will measure.

Not all long skills are long for the same reason. A conventional one-shot or community-authored skill can mix operational rules with background exposition, templates, examples, or even content unrelated to execution; deleting or deferring such material is often appropriate [1]. A multi-round evolved skill is different. Recent systems accept bounded edits only after rollout feedback or validation, or aggregate recurrent successful and failed trajectories into updates [2], [3]. Its additional text is therefore usually more knowledge-dense: even a singleton warning may encode an expensive failure that the agent learned to avoid. The main redundancy is less often *irrelevant topic* and more often *repeated representation*—

the same invariant copied into several branches, a workflow restated after each failure, or a general rule followed by progressively narrower exceptions. **This distinction changes the compression goal from content filtering to knowledge consolidating.**

The obvious response is prompt compression, but the abstraction is wrong. Prompt compressors typically decide which tokens are useful for a current query or likely answer [4]–[6], while skills should be reusable for all queries of certain tasks. More importantly, its meaning is not distributed uniformly across words. The name and description determine when the skill is selected; temporal phrases determine action order; tool arguments define valid calls; branch guards determine where a rule applies; and an output schema defines when the procedure is complete. A rare exception may be activated only once, yet deleting it can be more damaging than removing several paragraphs of rationale. Task-based validation is also not a complete solution. SkillReducer [1] demonstrates that structure-aware rewriting and progressive disclosure can reduce skill cost, and its feedback loop uses generated tasks to recover missed content. Such validation is valuable, but it makes compression expensive and couples the compressed results to compression-time tasks. A method can repeatedly repair the branches observed by that set while still losing an untested guard or output constraint. We therefore ask a stricter question: *Can a skill be shortened using only the structure already present in the skill, without observing tasks, rewards, trajectories, or verifiers?*

Our answer begins with a simple observation: a skill resembles a compact operating manual more than a generic prompt. It contains an **interface** (name, purpose, triggers, exclusions), a **procedure** (steps, branches, loops, fallbacks), **contracts** for tools and outputs, and **rules** whose scope can be global or branch-specific. This structure reveals redundancy that token importance cannot see. A rule repeated in every branch can be stated once before the branch. A repeated action sequence can be named once and reused. Several guarded variants can be written as one common rule plus explicit exceptions. Conversely, a unique requirement cannot be removed merely because it is short or rare.

Motivated by this observation, the key compression principle of SkillZip could be concluded as: **prefer the shortest faithful explanation of the skill**. Informally, the compressed representation pays once for every shared structure and pays separately only for genuine differences. This is the same intuition as replacing three copies of a code fragment by one function and three calls. Formally, it is an instance of minimum description length (MDL): minimize the cost of a compact skill contract plus the cost of the source details not explained by that contract, while requiring every extracted normative requirement to remain covered. We will further introduce the intuition before the formal objective in Section IV.

To broaden the practical adaptability of SkillZip, we design two modes. **One-shot SkillZip** compresses an existing evolved skill checkpoint with one structured extraction call followed by deterministic optimization. **Zip-on-Write** participates in self-evolution: when a new self-evolution patch arrives, SkillZip compares it only with compatible parts of the current compact contract, then either absorbs it, refines an existing rule, adds a new requirement, or triggers a local refactoring. Periodic repacking captures patterns that become reusable only after several updates.

SkillZip at a glance

Compress once: extract a typed contract, consolidate reusable rules and workflows, and render a shorter skill while preserving every covered requirement. **Maintain continually:** use Zip-on-Write to absorb each validated patch through local, scope-aware updates. **Evidence contract:** compression observes no benchmark tasks, rollouts, rewards, or behavioral verifier.

Our main contributions are:

- We formulate **evaluation-free skill compression** as contract representation tailored for evolved skills.
- We derive a **shortest faithful explanation objective** that unifies semantic sharing, scope lifting, workflow reuse, and exception encoding. A hard coverage constraint gives a rare-rule preservation guarantee independent of task frequency.
- We design **one-shot SkillZip** and **Zip-on-Write**. The first uses one structured extraction call followed by deterministic optimization; the second maintains a compact state during self-evolution and avoids replaying tasks or reparsing the full history.
- We perform comprehensive empirical evaluations showing that SkillZip delivers substantial gains in compression performance, robust generalizability, and low cost overhead.

II. Related Work

A. Self-Evolving Agents and Persistent Skills

Agents increasingly convert interaction experience into reflections, memories, programs, or reusable skills [7]–[9]. Systems such as ACE, SkillRL, and SkillClaw explicitly maintain and evolve persistent skill artifacts across tasks or users [3], [10], [11]; SkillRevise and SkillGrad use execution traces to diagnose and improve an existing skill [12], [13]. These works focus on how procedural knowledge is acquired. As the artifact grows, however, acquisition and maintenance become different problems: a useful new patch can still duplicate an old invariant or repeat an existing workflow. SkillZip addresses this consolidation problem without replaying the experiences that produced the skill.

B. Prompt and Context Compression

Prompt compression prunes or rewrites context using perplexity, query relevance, learned token labels, summarization, or latent representations [4]–[6], [14]–[16]. These techniques usually treat input as a sequence whose importance is estimated relative to a query, answer, or representative task distribution. Skills violate both assumptions: future tasks are unknown, and procedural meaning depends on typed relations such as "before", branch guards, tool arguments, and required output fields. Query-conditioned compression may be effective for a single request, but a reusable skill must retain requirements that no compression-time query activates. **SkillZip therefore compresses repeated procedural structure rather than low-salience tokens.**

C. Skill Compression and Efficient Execution

SkillReducer is the closest textual baseline. It minimizes routing descriptions with delta debugging, classifies body content, moves supplementary material to on-demand references, and uses faithfulness checks and task-based feedback [1]. It establishes that skill content should not be treated as homogeneous text and is especially well matched to ordinary skills that contain background, examples, and reference material. Multi-round evolved skills present a complementary regime: accepted updates are often operationally relevant, while redundancy appears as repeated constraints, overlapping workflows, and accumulated exceptions. In addition, SkillReducer's generated tasks and feedback loop participate directly in candidate selection. Repeatedly tuning an artifact on a finite evaluation sample can create sample-specific selection effects, a general issue in adaptive data analysis [17]. SkillZip removes this source of dependence by never exposing compression to tasks, and it keeps a single portable text skill rather than changing the loading architecture.

SKIM and TokMem encode procedural knowledge into learned soft or memory tokens [18], [19], while Skill-to-LoRA transfers a textual skill into model parameters [20]. Their representations can be compact, but they are model-dependent and less convenient to inspect, diff, or update during continual evolution. SkillZip uses a structured sidecar only while maintaining the artifact and renders an ordinary human-readable skill for deployment. Skill [21] and SkillRT [22] treat skills as executable or compilable artifacts. Their goals are reliable execution and portability rather than shortening an accumulated natural-language skill. Nevertheless, they support the premise underlying our representation: *a skill has an interface, control flow, and contracts, not merely topical text.*

D. Minimum Description Length

The minimum description length (MDL) principle selects the model that gives the shortest joint description of a model and the data it explains [23], [24]. Grammar-based methods such as SEQUITUR and Re-Pair similarly replace repeated subsequences with reusable rules [25], [26]. We adopt such an intuitive principle—*a shared explanation is useful when defining it once and referencing it is cheaper than repeating it*—and adapt it to typed procedural knowledge. Unlike sequence compression, our objective may not merge identical-looking clauses if they occur under incompatible guards, and it may not delete a unique rule even when that rule has zero repetition. The hard coverage is therefore as important as the skill length.

III. From Skill Text to a Compact Contract

A. Evaluation-Free Compression

Let S be one textual skill and $\widetilde{S}$ its compressed form. During compression, a method may read S, files referenced by S, and—in the continual setting—the sequence of skill patches. It may not access downstream tasks, execution trajectories, rewards, or behavioral verifiers. These resources are used only after compression to measure generalization. We seek three properties. First, $\widetilde{S}$ should be substantially shorter under the deployment tokenizer. Second, it should preserve what the skill *requires*, including rare conditions. Third, it should remain an ordinary text artifact that can be inspected, versioned, and used by different agent backbones.

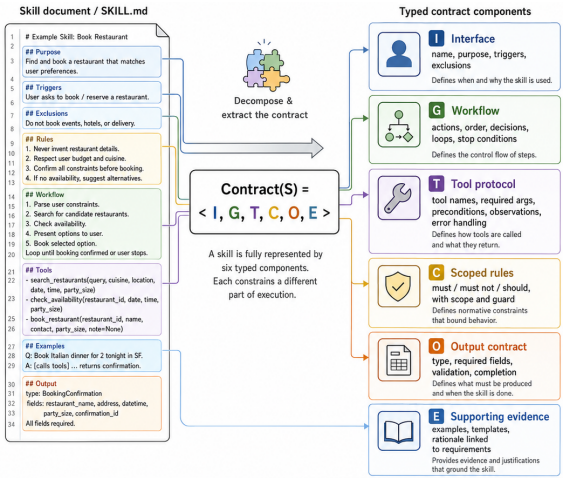

Fig. 2. **A skill is a typed contract.** Different text spans constrain different parts of execution. An example is removable only when every requirement it uniquely expresses is represented elsewhere.

The second requirement is deliberately structural. Without running tasks, we cannot prove that arbitrary natural language induces identical model behavior. We can, however, require the compressor to preserve every operational element it extracts from the source, and to retain ambiguous source spans verbatim. This makes the boundary of the guarantee explicit rather than hiding it behind a finite evaluation suite.

B. The Contract Hidden Inside a Skill

Figure 2 shows the representation used by SkillZip. We write the extracted contract as

$$C(S) = \langle I, G, T, C, O, E \rangle, \tag{1}$$

where:

- I is the **interface**: skill name, purpose, positive triggers, and exclusions;
- G is the **workflow**: actions, order, decisions, loops, fallbacks, and stop conditions;
- T is the **tool protocol**: tool names, required arguments, preconditions, expected observations, and error handling;
- C is the set of **scoped rules**: what the agent must, must not, or preferably should do, together with the scope and guard under which each rule applies;
- O is the **output contract**: response type, required fields, ordering, validation, and completion conditions; and
- E is **supporting evidence**: examples, templates, and rationale linked to the contract elements they express.

This decomposition is not cosmetic. It changes which compressions are safe. Two sentences about the same tool cannot merge if they require different arguments. A prohibition repeated in all branches may move to the common parent scope, but a prohibition attached to only one guarded branch may not. Two examples can be removed if they merely illustrate an

explicit output schema; an example that is the sole source of a required field must first be converted into an explicit output rule.

C. Typed Units, Scope, and Coverage

The parser maps source spans to typed units $a \in \mathcal{A}$. A unit records

$$a = (\tau, \sigma, g, m, p, \mathcal{P}), \tag{2}$$

where τ is its type, σ its scope, g an optional guard, m its modality, p its normalized content, and $\mathcal{P}$ the supporting source spans. A workflow unit additionally records incoming and outgoing edges; a tool unit records its argument signature; an output unit records field and validation information.

If a source span cannot be interpreted with sufficient confidence, it becomes a *locked residual*: it is copied verbatim and excluded from deletion. This conservative fallback is important in practice because the structural parser, rather than the later optimizer, is the main source of semantic uncertainty.

We write $a \preceq K$ when compact representation K covers unit a. Coverage can be direct, or structural: a branch-local copy can be covered by the same rule placed at the closest common ancestor; a repeated action sequence can be covered by a shared procedure whose expansion contains the original edges. Coverage is type-sensitive. A tool name does not cover its required arguments, and a general rule does not cover a conflicting guarded exception.

Definition III.1 (Contract coverage)**.** A compact representation K covers a parsed skill if

$$\forall a \in \mathcal{A}_{\text{req}}(S), \qquad a \preceq K, \tag{3}$$

where $\mathcal{A}_{\text{req}}$ contains interface conditions, workflow nodes and edges, tool requirements, scoped rules, and output requirements.

The next section asks which covering representation should be selected. The answer is not to merge everything similar, but to choose the representation that is shortest after accounting for definitions, references, exceptions, and unexplained residual text.

IV. Theoretical Analysis

Section III defines the *contract* that must survive compression. The remaining question is how to represent that contract compactly. Our intuition is the same as refactoring repeated code: *state shared structure once, reference it where needed, and keep only genuine differences explicit*. A shared abstraction is useful only when this representation is shorter than repeating all of its instances.

A. The Shortest Faithful Explanation

We represent a compressed skill by a library $\mathcal{K}$ of reusable contract elements and a residual $\mathcal{R}$ for unique, exceptional, or uncertain content. The library may contain shared rules, workflow fragments, tool contracts, output fields, and interface entries. The residual preserves explicit exceptions that cannot be safely normalized.

SkillZip selects the shortest representation that still covers every required contract unit:

$$\begin{aligned}(\mathcal{K}^*, \mathcal{R}^*) = \arg \min_{(\mathcal{K},\mathcal{R}) \in \mathcal{H}(S)} \Big[L(\mathcal{K}) + L(\mathcal{R} \mid \mathcal{K}) \Big] \\ \text{s.t.} \quad a \preceq (\mathcal{K}, \mathcal{R}), \qquad \forall a \in \mathcal{A}_{\text{req}}(S).\end{aligned} \tag{4}$$

Here, $\mathcal{H}(S)$ is the finite set of candidate representations proposed from the parsed skill, and $L(\cdot)$ measures the rendered token cost, including the overhead of definitions, references, and scope annotations. Equation (4) is an instance of minimum description length, but its operational meaning is simple: **compression may change how a requirement is written, but not whether it remains represented**. In particular, a unique requirement cannot be removed merely as it is short or unsupported by frequently sampled tasks.

B. One Objective, Four Skill-Specific Decisions

The same objective governs four forms of reusable structure.

- **Equivalent requirements:** paraphrases are represented once when a shared rule plus any residual differences is shorter than keeping separate copies.
- **Repeated rules across scopes:** a rule is moved to the nearest common scope only when it applies to every relevant path; conflicting branch-local behavior remains as an explicit exception.
- **Repeated workflows:** a recurring action sequence becomes a shared procedure only when the saved repetitions exceed the cost of defining and calling it.
- **Guarded variants:** related clauses may be represented as one common rule plus guarded deltas, but only when the exception structure is both faithful and shorter than listing the variants separately.

These decisions are therefore not independent rewrite heuristics. They are alternative ways of covering the same typed contract, compared under one length objective. The explicit cost tests and scope conditions are given in Appendix A.

C. Preservation and Continual Compression

Proposition IV.1 (Parsed-contract preservation)**.** *If every normative source span is represented by a typed unit or residual, any feasible solution of Eq.* (4) *preserves all extracted requirements.*

Prop. IV.1 follows directly from the hard coverage constraint. Its most important consequence is independent of the length model:

Corollary IV.2 (Rare-rule preservation)**.** *The preservation of a unique requirement does not depend on how often its branch appears in any compression-time task distribution.*

This is the key theoretical benefit of evaluation-free compression. A guard, tool argument, exception, or output field is protected because it belongs to the parsed contract, not because sampled tasks happen to activate it. The guarantee is intentionally limited to the extracted contract; uncertain spans are therefore kept verbatim rather than silently discarded. The objective also yields an efficient continual form. A new

patch usually changes only a small type–scope neighborhood of the current compact contract. If the patch creates no profitable abstraction that crosses this neighborhood, all other cost terms remain constant, so local optimization gives the same update as rerunning the batch objective. When several patches collectively create new cross-scope reuse, occasional global repacking restores the missed saving. Thus local updates provide efficiency, while repacking recovers long-range structure. Formal conditions and proofs are deferred to Appendix A.

V. SkillZip

Figure 3 summarizes the method. One-shot compression converts a skill into a compact contract and then renders it back to text. Zip-on-Write maintains the same contract while the agent evolves, so each new patch is consolidated before it becomes permanent.

A. One-Shot Compression

a) 1. Scan the SKILL.MD before using a model.: A deterministic scanner parses front matter, headings, nested lists, code blocks, tables, and file references. It copies the skill name and description as interface candidates, converts Markdown nesting into a preliminary scope tree, and records stable source-block identifiers. Numbered lists and temporal markers provide high-confidence workflow hints. This pass reduces the amount of structure that the language model must infer and makes every later compression decision traceable to source skill text.

b) 2. Recover the typed contract once.: A schema-constrained model receives the numbered blocks and returns the contract in Eq. (1): interface entries, workflow nodes and edges, tool calls and required arguments, scoped rules with modality and guard, output fields, and links from examples to the requirements they specify. Every extracted unit must cite source blocks. The host rejects unsupported citations, polarity mismatches, unknown tool names, and invalid workflow references. Ambiguous spans are placed in the locked residual. The extractor is deliberately not asked to compress. Separating interpretation from optimization has two benefits: contract recovery can be evaluated against human annotations, and the optimization is deterministic once the extracted units are fixed.

c) 3. Propose only type-compatible reuse.: Candidate generation uses hard structural blocking: (a) interface entries compare with the same trigger or exclusion role; (b) rules compare when modality, predicate family, and scope ancestry are compatible; (c) tool units require the same tool and compatible argument signatures; (d) output units compare within the same response type and field namespace; (e) and workflow reuse is proposed from repeated guarded action sequences with identical entry and exit behavior. Exact matches are found by hashing. Near duplicates are retrieved with an embedding index and verified by a frozen relation checker that predicts equivalence, implication, conflict, or unrelatedness. Conflicts create exception candidates rather than merge candidates.

Algorithm 1 One-Shot SkillZip

Require: Skill S and audit flag v
Ensure: Compressed skill $\widetilde{S}$

$B \leftarrow \textsc{Scan}(S)$
$(\mathcal{A}, R) \leftarrow \textsc{ExtractContract}(B)$
$H \leftarrow \textsc{ProposeReuse}(\mathcal{A})$
$K \leftarrow \textsc{MinCostCover}(\mathcal{A}, R, H)$
$\widetilde{S} \leftarrow \textsc{Render}(K)$
if v **then**
  $\widehat{K} \leftarrow \textsc{Parse}(\widetilde{S})$
  $M \leftarrow \textsc{ContractDiff}(K, \widehat{K})$
  $\widetilde{S} \leftarrow \textsc{RestoreMissingSpans}(\widetilde{S}, M)$
return $\widetilde{S}$

d) 4. Select the shortest covering explanation.: Each candidate h receives a saving

$$\text{save}(h) = L(\text{separate form}) - L(\text{form using } h). \quad (5)$$

Non-positive candidates are discarded. Equivalent units are clustered after conflict filtering. Rule placement is solved by dynamic programming over the scope tree. Repeated workflow candidates are selected by non-overlapping weighted packing, using saving per covered token as the greedy order and pairwise exchange as a refinement. After each selection, the optimizer checks that all source units remain covered. This step is where the running example is simplified. The two branch-local copies of "never overwrite the input" are represented by one rule at their common parent because Eq. (8) is satisfied. The validate step is shared only if its definition and calls are shorter than the copies. The JSON example is removed only after its fields are covered by the output contract.

e) 5. Render with fixed templates and audit structurally.: The renderer produces a normal skill with a concise purpose and triggers, global rules, a numbered workflow, nested guarded branches, explicit tool requirements, and an output checklist or schema. A shared workflow is named only when references save tokens; otherwise it remains inline. Exceptions are placed after their base rule. An optional structural audit reparses only the compressed skill and compares it with the selected contract. If a trigger, guard, workflow edge, tool argument, polarity, or output field is missing, SkillZip restores the shortest source span that covers it and locks that span.

B. Continual Compression: Zip-on-Write

A self-evolving agent usually produces a small patch rather than a complete rewrite. SkillZip stores a sidecar, `skillzip.json`, containing the current contract, source provenance, scope tree, workflow graph, and candidate indices. The rendered `SKILL.MD` remains the only artifact loaded by the agent.

For every patch unit, the updater compares four interpretations under the same objective:

- `ABSORB`: the patch restates an existing requirement and adds no new contract content;
- `REFINE`: the patch adds a guard, tool argument, validation, or explicit exception to an existing unit;

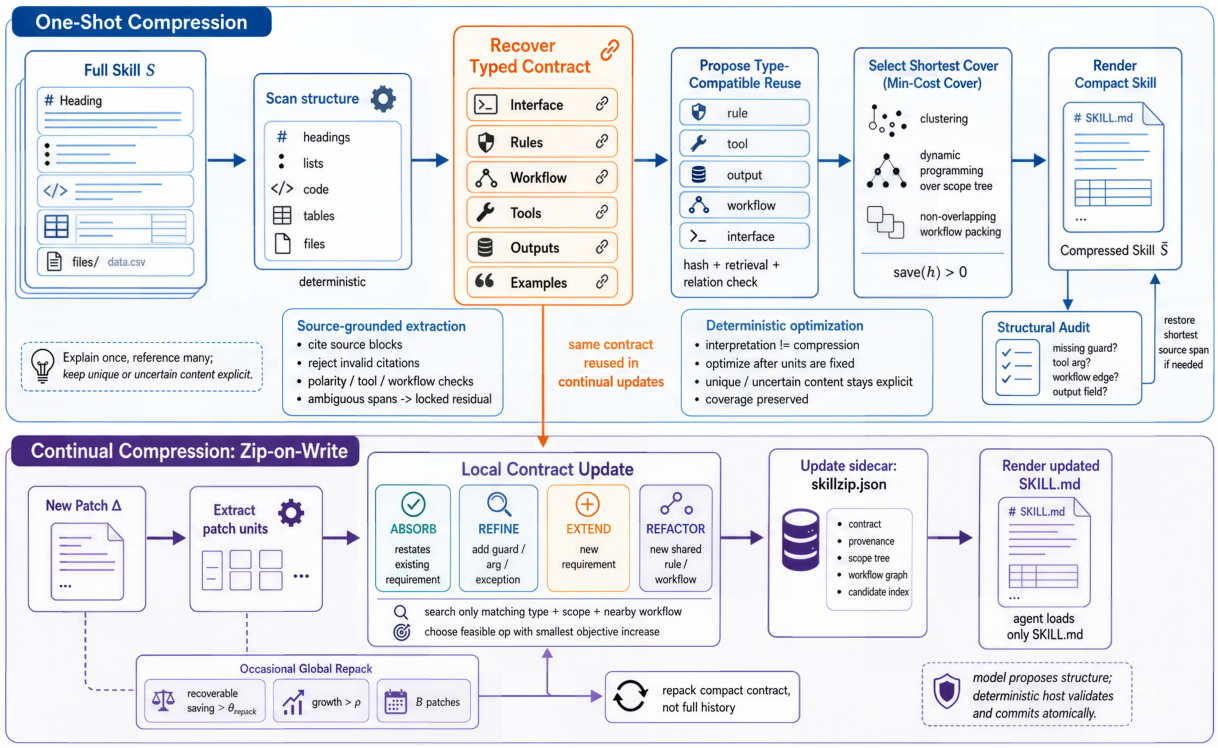

Fig. 3. **Overview of SkillZip.** One-shot compression first recovers the skill contract, then applies the "explain once, reference many" principle to repeated rules and workflows, while unique and uncertain content remains explicit. Zip-on-Write compares each patch with the affected contract neighborhood and performs occasional repacking when reuse accumulates across patches.

- `EXTEND`: the patch introduces a genuinely new requirement; and
- `REFACTOR`: the patch makes a shared rule or workflow newly worthwhile.

The host selects the feasible operation with the smallest increase in Eq. (4). Candidate search is restricted to the matching type, current scope, ancestor scopes, and adjacent workflow nodes. Thus a patch with d extracted units compares against $O(dk)$ retrieved candidates rather than the complete history.

Local updates may miss reuse that becomes profitable only after several patches. The sidecar therefore tracks approximate counts for rule families and action n-grams. A global repack is triggered when the estimated recoverable saving exceeds θ_{repack}, the contract grows by more than ρ since the last repack, or B patches have arrived. Repacking operates on the compact contract rather than all historical prose.

Algorithm 2 in Appendix A summarizes the continual path explicitly. Crucially, Δ_t is frozen before compression: the evolver decides what knowledge is learned, while SkillZip decides only how that knowledge is represented. Absorb is feasible only when the patch adds no uncovered contract unit; Refine preserves the old unit and records its new guard, argument, validation, or exception; Extend adds a new required unit; and Refactor changes representation without changing coverage. No operation is accepted because it improves a task score. Appendix B specifies the schema, cache keys, candidate algorithms, prompts, transaction protocol, and recovery path needed for a reproducible implementation.

a) Compression as a skill.: Skill compression can also be packaged as a skill. The model receives the new patch and a retrieved slice of the sidecar, then proposes one of the four operations with source citations. A deterministic host validates the schema, recomputes the saving, enforces coverage, writes a transaction log, and atomically replaces the skill only after rendering succeeds. The model proposes structure but never directly mutates persistent state.

VI. Experiments

We evaluate SkillZip along the two claims that motivate its design: self-evolution introduces substantially more *surface text* than new procedural knowledge, and this redundancy can be removed without consulting downstream evaluations. The section first defines the controlled and model-backed protocols, then answers five research questions: **RQ1** quantifies skill growth during self-evolution; **RQ2** studies the compression–fidelity trade-off; **RQ3** measures compression cost; **RQ4** evaluates the compressed skills' generalizability of SkillZip against the baseline; and **RQ5** evaluates continual Zip-on-Write compression.

A. Experimental Setup

a) Models and Benchmarks.: We evaluate three agent model backbones: Qwen3.7-Max, Qwen3.6-Plus, and Kimi K2.6 [27]–[29]. The two Qwen models represent different capability tiers within the same model family, while Kimi K2.6 provides a cross-family evaluation. For every model–benchmark pair, all skill conditions use the same model

snapshot, agent scaffold, system prompt, tool definitions, maximum interaction budget, and decoding configuration. We consider three benchmarks with complementary procedural requirements. **BFCL-v4 Web Search** [30], [31] evaluates multi-step web retrieval and reasoning through standardized search and webpage-access tools. **LiveMathematicianBench** [32] contains theorem-grounded multiple-choice problems derived from recent mathematical research and tests precise reasoning about assumptions, quantifiers, equivalence relations, and boundary conditions. **SpreadsheetBench** [33] evaluates spreadsheet manipulation on realistic user requests and workbook files.

b) Skill construction.: For each model–benchmark pair, we begin with a manually authored *human skill* that specifies the task procedure and output requirements. We then apply SkillOpt [2] to improve this seed skill using the designated evolution split. SkillOpt iteratively converts execution feedback into bounded edits of the skill document. The resulting skill is frozen after evolution and used as the common input to all compression methods. The evolution split is disjoint from the final benchmark test set. Thus, test instances cannot affect either the knowledge acquired during skill evolution or the subsequent compression decisions. Note that for one-shot SkillZip, we use a single fixed compressor model (Qwen3.7-max) across all skills, whereas for continual Zip-on-Write the agent's backbone model compresses its own skill. In both modes, the model is invoked only for schema-constrained contract extraction and relation/merge adjudication (greedy decoding, temperature 0); the minimum-cost covering and rendering are deterministic.

c) Baselines.: We compare SkillZip with the following five Baselines: (1) **No Skill**: the backbone model is evaluated without any task-specific skill; (2) **Human Skill**: the original manually authored seed skill is provided to the model; (3)**Evolved Skill**: the complete, uncompressed skill produced by SkillOpt is provided to the model; (4)**SkillReducer**: SkillReducer [1] is applied once to the frozen evolved skill. The uncompressed evolved skill serves as the fidelity reference for the two compression methods. SkillReducer is the primary compression baseline because it is the closest prior method that directly optimizes textual agent skills, rather than generic prompts or interaction histories. We run SkillReducer using its native structure-aware pipeline. SkillZip receives exactly the same evolved skill but does not access benchmark tasks, execution trajectories, rewards, or behavioral verifiers during compression.

B. RQ1: Skill Growth in Self-Evolution?

Figure 4 shows that skill length increases monotonically with self-evolution rounds across all benchmarks. By Round 5, the skills reach approximately 5.6×, 3.1×, and 6.7× their initial sizes on BFCL-V4, LiveMath, and SpreadsheetBench, respectively, with an average growth of about 5.2×. The growth persists across domains and is especially pronounced for tasks that continually accumulate tool-use procedures, failure corrections, and output constraints. Although each update may be locally useful, repeated rules, overlapping workflows, and increasingly specific exceptions accumulate without global consolidation, producing substantial *skill bloat* and increasing the context cost of every subsequent invocation.

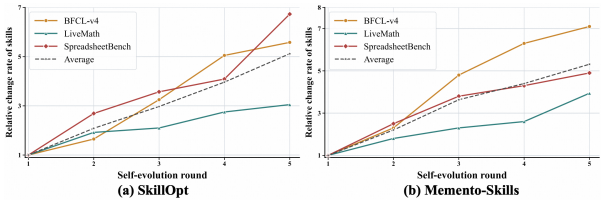

Fig. 4. **The skill growth tendency with respect to agent self-evolving progress for SkillOpt [2] and Memento-Skills [34].**

Takeaway

Self-evolution expands skills to more than 5× **their initial length on average**, making compression necessary to prevent procedural knowledge from becoming an increasing context burden.

C. RQ2: Can SkillZip Preserve Skill Fidelity?

Tab. I compares the uncompressed SkillOpt-evolved skills with their compressed ones and other skill settings. The evolved skill outperforms the no-skill and human-skill conditions in eight of nine settings, confirming that multi-round evolution generally accumulates useful procedural knowledge. SkillZip retains this knowledge while achieving compression rates of 27.1%–36.9% (**31.2% on average**). Its macro-average score is 0.577, slightly exceeding the uncompressed evolved skill at 0.570, and it matches or improves the evolved skill in five of nine settings. These results suggest that consolidating repeated rules, scopes, and workflows can reduce skill length without systematic behavioral degradation, even though SkillZip uses no tasks, rollouts, or verifiers during compression.

TABLE I
COMPRESSED SKILL PERFORMANCE AND AVERAGE SKILL COMPRESSION RATE. THE BEST TASK PERFORMANCE IS SHOWN IN **BOLD**. COMPRESSION RATE (C-RATE) IS MEASURED RELATIVE TO THE CORRESPONDING UNCOMPRESSED EVOLVED SKILL.

Model	Skill Condition	BFCL-V4 ↑	LiveMath ↑	Spreadsheet ↑	Avg. C-Rate↑
Qwen-3.7-Max	No Skill	0.833	0.425	0.432	–
	Human Skill	0.848	0.396	0.436	–
	Evolved Skill	**0.869**	**0.474**	0.525	0%
	SkillReducer	0.828	0.428	**0.538**	10.5%
	SkillZip	0.863	0.472	0.519	**27.1%**
Qwen-3.6-Plus	No Skill	0.623	0.417	0.379	–
	Human Skill	0.641	0.405	0.457	–
	Evolved Skill	0.685	0.392	0.472	0%
	SkillReducer	0.621	0.385	0.484	3.6%
	SkillZip	**0.694**	**0.435**	**0.491**	**29.7%**
Kimi-K2.6	No Skill	0.714	0.402	0.415	–
	Human Skill	0.708	0.362	0.473	–
	Evolved Skill	**0.772**	0.433	0.506	0%
	SkillReducer	0.732	0.384	0.497	13.4%
	SkillZip	0.747	**0.457**	**0.513**	**36.9%**

Compared with SkillReducer [1], SkillZip achieves both higher compression (31.2% vs. 9.2% on average) and higher task performance (0.577 vs. 0.544). *This difference reflects the distinct compression regimes targeted by the two methods. SkillReducer is well suited to an initial debloating and quality-control pass over heterogeneous public skills, which may*

contain verbose background, redundant examples, or content unrelated to execution. In contrast, self-evolved skills are typically knowledge-dense because their updates arise from execution feedback; their main redundancy lies in repeatedly stated constraints, overlapping scopes, and copied workflows. Consequently, **structural consolidation is better aligned with evolved-skill compression than content filtering or deferral**.

Takeaway

SkillZip compresses evolved skills by **31.2% on average** while preserving or improving their overall performance; SkillReducer is more naturally positioned as a first-pass debloating and quality-control method for general public skills.

TABLE II
COMPRESSION OVERHEAD OF SKILLZIP AND SKILLREDUCER.

Method	Dataset	Average Time ↓	LLM Calls ↓	Rollouts ↓
SkillZip	LiveMath	**207 s**	4	**0**
SkillReducer	LiveMath	1331 s	**3**	40
SkillZip	Spreadsheet	**332 s**	5	**0**
SkillReducer	Spreadsheet	1082 s	**3**	80
SkillZip	BFCL-V4	**318 s**	8	**0**
SkillReducer	BFCL-V4	587 s	**3**	40

Note: "LLM calls" counts only calls to the compressor model. SkillReducer additionally consumes 40–80 validation rollouts per compression (each requiring at least one agent call). Because some rollouts hit a warm evaluation cache, the reported time is an optimistic lower bound.

D. RQ3: Compression Efficiency

To assess the compression cost of SkillZip compared to SkillReducer, we compute the offline one-shot cost with SkillReducer on all benchmarks under the same execution environment. SkillZip completes compression faster on all three datasets, reducing average time cost from 1331 to 207 seconds on LiveMath, from 1082 to 332 seconds on Spreadsheet, and from 587 to 318 seconds on BFCL-V4. Averaged across datasets, SkillZip requires 286 seconds, corresponding to a **3.5× speedup**.

SkillReducer uses fewer direct compression-model calls, but additionally requires 40–80 task rollouts for candidate validation and repair. In contrast, SkillZip uses several structured LLM calls but **requires no task rollout on any dataset**. The results indicate that environment interaction, rather than the number of compression calls alone, dominates the end-to-end cost of evaluation-guided compression. Consequently, the evaluation-free design of SkillZip substantially reduces latency while avoiding potential overfitting and dependence on executable tasks and behavioral verifiers.

Takeaway

SkillZip achieves a **3.5× average speedup** over SkillReducer while requiring **zero task rollouts**, highlighting the efficiency.

E. RQ4: Cross-Model Generalization

Fig. 6 evaluates whether a skill compressed from one source model can be executed by a different target model. On LiveMath, SkillZip achieves an overall retention of **0.97**, compared with 0.91 for SkillReducer. Since their same-model results are comparable, the improvement mainly comes from off-diagonal source–target pairs, suggesting that preserving explicit rules, guards, and output constraints produces a more model-independent skill representation.

Takeaway

SkillZip transfers compressed skills across agent backbones without target-specific evaluation, improving retention on LiveMath and remaining comparable to SkillReducer on BFCL-V4.

F. RQ5: Continual Zip-on-Write Compression

Fig. 5 evaluates Zip-on-Write, the continual mode of SkillZip, inside a 16-round self-evolution loop on LiveMath with three backbones. Without compression, the SkillOpt evolver exhibits the bloat pathology consistently across all three models: the length of skill grows monotonically to 2.5×, 3.1×, and 3.7× its seed length, respectively. Activating Zip-on-Write from round 1 *bounds* this growth for the entire trajectory, capping the skill at roughly 1.6×–1.9× across all three models—a 38%–50% reduction relative to the uncompressed endpoint—because each write is immediately absorbed into the typed contract library and periodically repacked.

Two further observations yield the practical guidance. First, activation time matters: switching compression on only at round 8 recovers part of the accumulated redundancy but never catches up with the early-activation trajectory (e.g., 2.6× vs. 1.9× on Kimi-k2.6), showing that redundancy is cheaper to prevent than to remove. Second, compression does not trade accuracy for compactness: the final held-out test accuracy of the round-1 configuration matches or slightly exceeds the uncompressed skill on all three backbones.

Takeaway

Continual compression should be on from the start of evolution—it *keeps the skill within an obviously smaller length factor of the uncompressed one at no accuracy cost*, whereas *a later activation of compression only partially undoes bloat that has already compounded.*

VII. Conclusion

Self-evolving agents need a mechanism for forgetting repetition without forgetting procedure. SkillZip treats a skill as a typed contract and compresses it using a simple principle: explain shared structure once, reference it where needed, and keep genuine differences explicit. The resulting shortest-faithful-explanation objective is evaluation-free, protects rare requirements through hard coverage, and unifies rule sharing, scope placement, workflow reuse, and exception handling. One-shot compression requires one structured extraction call followed by deterministic optimization; Zip-on-Write integrates the same principle into continual skill evolution. The proposed experiments separate structural correctness, held-out behavior, evaluation-set overfitting, and cost, providing a practically reliable skill compression method tailored for self-evolving agents.

Fig. 5. Skill length (by $N\times$) during self-evolution on LiveMath for three agent backbones. Each panel compares no compression against ZIP-ON-WRITE continual compression activated at round 8 and at round 1; legends report the final test accuracy.

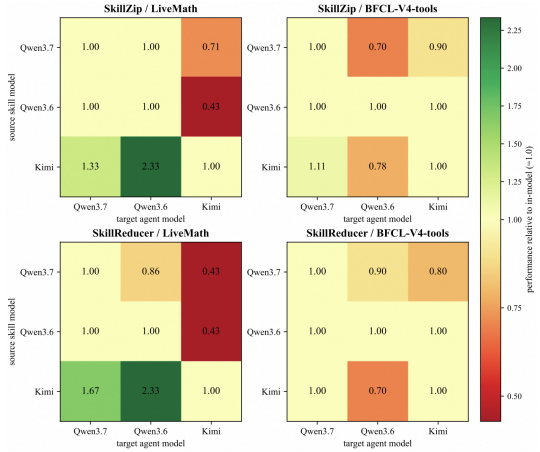

Fig. 6. Cross-model generalization of compressed skills on LiveMath and BFCL-V4. Rows denote the source model whose evolved skill is compressed, and columns denote the target model executing the compressed skill. Diagonal cells represent same-model deployment, while off-diagonal cells measure cross-model transfer.

REFERENCES

[1] Y. Gao, Z. Li, Y. Yuan, Z. Ji, P. Ma, and S. Wang, "Skillreducer: Optimizing llm agent skills for token efficiency," *arXiv preprint arXiv:2603.29919*, 2026.

[2] Y. Yang, Z. Gong, W. Huang, Q. Yang, Z. Zhou, Z. Huang, Y. Li, X. Gao, Q. Dai, B. Liu, K. Qiu, Y. Yang, D. Chen, X. Yang, and C. Luo, "Skillopt: Executive strategy for self-evolving agent skills," *arXiv preprint arXiv:2605.23904*, 2026.

[3] Z. Ma, S. Yang, Y. Ji, X. Wang, Y. Wang, Y. Hu, T. Huang, and X. Chu, "Skillclaw: Let skills evolve collectively with agentic evolver," *arXiv preprint arXiv:2604.08377*, 2026.

[4] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, "Llmlingua: Compressing prompts for accelerated inference of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13 358–13 376.

[5] H. Jiang, Q. Wu, X. Luo, D. Li, C.-Y. Lin, Y. Yang, and L. Qiu, "LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1658–1677. [Online]. Available: https://aclanthology.org/2024.acl-long.91/

[6] Z. Pan, Q. Wu, H. Jiang, M. Xia, X. Luo *et al.*, "Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024.

[7] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *Transactions on Machine Learning Research*, 2023.

[8] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[9] H.-a. Gao, J. Geng, W. Hua, M. Hu, X. Juan *et al.*, "A survey of self-evolving agents: On path to artificial super intelligence," *arXiv preprint arXiv:2507.21046*, 2025.

[10] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li, U. Thakker, J. Zou, and K. Olukotun, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025.

[11] P. Xia, J. Chen, H. Wang, J. Liu, K. Zeng, Y. Wang, S. Han, Y. Zhou, X. Zhao, H. Chen, Z. Zheng, C. Xie, and H. Yao, "Skillrl: Evolving agents via recursive skill-augmented reinforcement learning," *arXiv preprint arXiv:2602.08234*, 2026.

[12] Y. Liu, Z. Su, L. Xie, Y. Zhang, Q. Zong, J. Guo, Z. Xie, Y. Ji, Y. Yim, H. Luo, X. Ren, R. Chenyu, H. Li, and Y. Song, "Skillrevise: Improving llm-authored agent skills via trace-conditioned skill revision," *arXiv preprint arXiv:2606.01139*, 2026.

[13] H. Wang, Y. Lan, B. Cao, L. Lin, and J. Chen, "Skillgrad: Optimizing agent skills like gradient descent," *arXiv preprint arXiv:2605.27760*, 2026.

[14] Y. Li, B. Dong, F. Guerin, and C. Lin, "Compressing context to enhance inference efficiency of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 6342–6353.

[15] J. Mu, X. L. Li, and N. Goodman, "Learning to compress prompts with gist tokens," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 19 327–19 352.

[16] F. Xu, W. Shi, and E. Choi, "Recomp: Improving retrieval-augmented lms with compression and selective augmentation," in *International Conference on Learning Representations*, 2024.

[17] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth, "Generalization in adaptive data analysis and holdout reuse," in *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[18] C. Wang, W. Su, Q. Ai, Y. Tang, R. Qiao, X. Li, M. Zhang, and Y. Liu, "Adaptive multi-resolution procedural knowledge compression for large language models," *arXiv preprint arXiv:2606.12203*, 2026.

[19] Z. Wu, Y. Hao, and L. Mou, "Tokmem: Tokenized procedural memory for large language models," *arXiv preprint arXiv:2510.00444*, 2025.

[20] T. Zhang and Z. Qi, "Skill-to-lora: From using skills to learning behaviors for token-efficient llm agents," *arXiv preprint arXiv:2606.16769*, 2026.

[21] X. Zhang, M. Gao, Y. Zhao, X. Tan, Y. Yao, F. Wang, Y. Wang, Dingsiyi, and T. Yang, "Formal skill: Programmable runtime skills for efficient and accurate llm agents," *arXiv preprint arXiv:2605.19604*, 2026.

[22] L. Chen, E. Feng, Y. Xia, and H. Chen, "Skillrt: Compiling skills for efficient execution everywhere," *arXiv preprint arXiv:2604.03088*, 2026.

[23] P. D. Gr"unwald, *The Minimum Description Length Principle*. MIT Press, 2007.

[24] E. Galbrun, "The minimum description length principle for pattern mining: A survey," *Data Mining and Knowledge Discovery*, vol. 36, no. 5, pp. 1679–1727, 2022.

[25] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[26] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[27] Qwen Team, "Qwen3.7: The agent frontier," Qwen Research Blog, May 2026, introduces the Qwen3.7 series, including Qwen3.7-Max. Accessed: 2026-07-23. [Online]. Available: https://qwen.ai/blog?id=qwen3.7

[28] ——, "Qwen3.6-Plus: Towards real world agents," Qwen Research Blog, Apr. 2026, accessed: 2026-07-23. [Online]. Available: https://qwen.ai/blog?id=qwen3.6

[29] Moonshot AI, "Kimi K2.6: Advancing open-source coding," Kimi Technical Blog, 2026, accessed: 2026-07-23. [Online]. Available: https://www.kimi.com/blog/kimi-k2-6

[30] S. G. Patil, H. Mao, F. Yan, C. C.-J. Ji, V. Suresh, I. Stoica, and J. E. Gonzalez, "The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models," in *Proceedings of the 42nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 267. PMLR, 2025, pp. 48 371–48 392.

[31] Gorilla Team, "BFCL V4: Agentic part 1—web search," Berkeley Function Calling Leaderboard Technical Blog, 2025, accessed: 2026-07-23. [Online]. Available: https://gorilla.cs.berkeley.edu/blogs/15_bfcl_v4_web_search.html

[32] L. He, Q. Yu, H. Dong, B. Liao, X. Xu, M. Goldblum, J. Bian, and N. Mesgarani, "LiveMathematicianBench: A live benchmark for mathematician-level reasoning with proof sketches," *arXiv preprint arXiv:2604.01754*, 2026.

[33] Z. Ma, B. Zhang, J. Zhang, J. Yu, X. Zhang, X. Zhang, S. Luo, X. Wang, and J. Tang, "Spreadsheetbench: Towards challenging real world spreadsheet manipulation," in *Advances in Neural Information Processing Systems*, vol. 37, 2024.

[34] H. Zhou, S. Guo, A. Liu, Z. Yu, Z. Gong, B. Zhao, Z. Chen, M. Zhang, Y. Chen, J. Li, R. Yang, Q. Liu, X. Yu, J. Zhou, N. Wang, C. Sun, and J. Wang, "Memento-skills: Let agents design agents," 2026. [Online]. Available: https://arxiv.org/abs/2603.18743

Appendix A
Additional Theoretical Details

Appendix roadmap

The public appendix records the decision-specific cost tests and proofs behind the shortest-faithful-explanation objective, the implementation and continual-update protocol, the full contract schema and prompts, the experimental protocol, and additional analysis of compression behavior. It is organized as a reproducibility companion to the main paper rather than as a space-constrained supplement.

A. Practical Length Model

The abstract length in Eq. (4) is instantiated as

$$L(x) = |\,\mathrm{Render}(x)|_{\mathrm{tok}} + \gamma_{\mathrm{def}}(x) + \gamma_{\mathrm{ref}}(x) + \gamma_{\mathrm{scope}}(x), \tag{6}$$

where the first term is the rendered token length and the remaining terms charge for defining a named abstraction, referencing it, and expressing its scope. These charges prevent degenerate solutions that introduce many tiny abstractions whose notation costs more than the text they replace.

B. Decision-Specific Cost Tests

a) Equivalent requirements.: Suppose clauses x_1 and x_2 can be represented by a common typed unit z. Sharing is selected when

$$L(z) + L(x_1 \mid z) + L(x_2 \mid z) < L(x_1) + L(x_2). \tag{7}$$

For true paraphrases, the residual terms are nearly empty. A polarity change, guard difference, tool-argument difference, or output-field difference must be encoded in the residual and can make sharing unprofitable.

b) Scope lifting.: Let rule c occur in child scopes $s_1, \ldots, s_r$. It may be moved to their closest common ancestor u only if every relevant path from u requires the rule and all local conflicts are encoded as exceptions. Among feasible placements, lifting is selected when

$$L(c@u) + rL(\textsf{scope-ref}) < \sum_{i=1}^{r} L(c@s_i). \tag{8}$$

c) Workflow reuse.: Let workflow fragment q occur r non-overlapping times. A shared procedure is useful when

$$L(\mathrm{def}(q)) + rL(\mathrm{call}(q)) < rL(q). \tag{9}$$

The threshold adapts to fragment length, the number of occurrences, and the definition/call overhead.

d) Common rule with exceptions.: Suppose guarded clauses $c_i@g_i$ share a common core c. SkillZip compares

$$L(c) + \sum_i L(\delta_i) \qquad \text{against} \qquad \sum_i L(c_i@g_i), \tag{10}$$

where δ_i records the guarded difference. The common representation is selected only when it is shorter and every exception remains attached to the rule it modifies.

C. Proofs and Additional Guarantees

Proof of Proposition IV.1. Feasibility requires every $a \in \mathcal{A}_{\mathrm{req}}(S)$ to be covered. Coverage may be direct, realized by a scope-safe shared rule, realized by a shared workflow whose expansion contains the original nodes and edges, or preserved verbatim in the residual. Removing the only representation of any required unit violates the constraint in Eq. (4); therefore, no feasible solution can discard it. □

Proposition A.1 (Reuse threshold)**.** *For a repeated structure q, introducing a shared definition is beneficial if and only if the saved repetition cost exceeds the definition and reference overhead.*

Proof. The separate form costs $rL(q)$, whereas the shared form costs $L(\mathrm{def}(q)) + rL(\mathrm{call}(q))$. The shared form reduces the objective exactly when Eq. (9) holds. □

Proposition A.2 (Local update equivalence)**.** *Let K_{t-1} be the compact contract before patch Δ_t. If the patch introduces no profitable abstraction whose occurrences span both the affected and unaffected regions, minimizing Eq.* (4) *over the affected type–scope neighborhood yields the same update as rerunning the batch optimizer on the full contract.*

Proof. Under the stated condition, every candidate whose feasibility or cost changes is contained in the affected neighborhood. All candidates and cost terms outside that neighborhood are identical before and after the patch and therefore contribute the same constant to the local and global objectives. Both optimizations consequently select the same change. □

Algorithm 2 Zip-on-Write Continual Compression

Require: State Z_{t-1}, accepted patch Δ_t, repack policy η
Ensure: Updated state Z_t and rendered skill $\widetilde{S}_t$

$B_t \leftarrow \textsc{ScanPatch}(\Delta_t)$
$(\mathcal{A}_t, R_t) \leftarrow \textsc{ExtractPatch}(B_t)$
$N_t \leftarrow \textsc{RetrieveCompatible}(Z_{t-1}, \mathcal{A}_t)$
$H_t \leftarrow \textsc{ProposeOps}(\mathcal{A}_t, N_t)$ ▷ absorb/refine/extend/refactor
$Z' \leftarrow \textsc{LocalMinCostUpdate}(Z_{t-1}, \mathcal{A}_t, R_t, H_t)$
$\textsc{AssertCoverage}(Z', \mathcal{A}_t)$
$\textsc{UpdateReuseStatistics}(Z')$
if $\textsc{RepackDue}(Z', \eta)$ **then**
    $U' \leftarrow \textsc{Units}(Z')$
    $R' \leftarrow \textsc{Residuals}(Z')$
    $H \leftarrow \textsc{ProposeReuse}(U')$
    $Z' \leftarrow \textsc{MinCostCover}(U', R', H)$
$\widetilde{S}_t \leftarrow \textsc{Render}(Z')$
if $\textsc{AuditDue}(Z')$ **then**
    $(Z', \widetilde{S}_t) \leftarrow \textsc{StructuralAuditAndRestore}(Z', \widetilde{S}_t)$
$Z_t \leftarrow \textsc{AtomicCommit}(Z', \widetilde{S}_t)$
return $(Z_t, \widetilde{S}_t)$

The condition of Proposition A.2 can fail when several patches jointly create a repeated workflow or a rule shared across previously unrelated scopes. This motivates periodic global repacking over the compact contract. Repacking is not required for correctness—coverage remains enforced after every patch—but it can recover savings invisible to purely local updates.

D. Boundary of the Guarantee

The preservation guarantee is relative to the contract produced by the parser; it does not establish that arbitrary natural language has been interpreted perfectly or that two rendered skills induce identical behavior for every language model. SkillZip makes this boundary explicit through source provenance, parser confidence, locked residuals, and the optional independent structural audit. Its intended conservative failure mode is under-compression, not silent deletion of uncertain requirements.

Appendix B
Implementation Details

This appendix specifies an implementation that can be translated directly into code. The intended repository separates parsing, optimization, rendering, and evaluation so the evaluation-free claim can be audited from file access and logs.

A. Repository and Command-Line Interface

```
skillzip/
  scanner.py # Markdown blocks + stable IDs
  extract.py # schema-constrained contract parse
  relations.py # typed retrieval + NLI checks
  workflow.py # graph and repeated-sequence mining
  optimize.py # min-cost covering selection
  render.py # deterministic text templates
  audit.py # compressed-text contract diff
  online.py # Zip-on-Write + transaction log
  schemas/skillzip.schema.json
  prompts/{extract,patch,audit}.txt
  configs/{default,efficient}.yaml
```

```
# one-shot
skillzip compress SKILL.md \
  --config configs/default.yaml \
  --state skillzip.json \
  --output SKILL.compact.md

# one continual update
skillzip update skillzip.json PATCH.md \
  --output SKILL.md

# evaluation-free structural check
skillzip audit SKILL.compact.md skillzip.json

# explain each accepted/rejected abstraction
skillzip inspect skillzip.json --show-savings
```

The CLI writes through temporary files and replaces the persistent state only after schema validation, coverage validation, and rendering succeed.

B. Deterministic Scanning and Stable Provenance

The scanner emits blocks with fields `id`, `kind`, `heading_path`, `text`, and `line_range`. The identifier is a hash of normalized text and ancestor headings; line numbers are stored separately so unrelated insertions do not invalidate provenance. Code fences and tables remain atomic blocks because splitting them can destroy a template or schema.

A scope path is an array such as

```
["root", "workflow", "if-validation-fails"]
```

Markdown nesting supplies initial scopes. Explicit guards create child scopes even when no heading is present. Promotion to a wider scope requires cited language such as “always”, “for every request”, or structural repetition across all relevant children.

C. Relation Checking Pipeline

Hard compatibility keys are applied before embeddings:

$$(\text{type}, \text{modality}, \text{tool/output namespace}, \text{scope family}). \tag{11}$$

Exact normalized units merge by hash. Remaining units are embedded once in a separate index per key. The top-k candidates are passed to a frozen cross-encoder that returns equivalence, left implication, right implication, conflict, or unrelated. Only equivalence and implication create sharing candidates. Conflict creates an exception edge; low-confidence pairs remain separate.

Cache keys include normalized texts, types, scope signatures, model and revision, thresholds, and prompt hashes. This supports deterministic reruns and avoids repeated relation calls in Zip-on-Write.

D. Practical Length Model

For a rendered unit or abstraction x,

$$\widehat{L}(x) = |\operatorname{Render}(x)|_{\text{tok}} + \lambda_d n_{\text{def}}(x) + \lambda_r n_{\text{ref}}(x) + \lambda_s d(x), \tag{12}$$

where $d(x)$ is scope depth. The default λ values equal the actual token cost of the corresponding template delimiters. They are not fitted on downstream tasks. Every candidate log stores separate before, definition, reference, exception, residual, and after costs.

```
{
  "candidate": "workflow:validate-repair",
  "before_tokens": 94,
  "definition_tokens": 32,
  "reference_tokens": 17,
  "exception_tokens": 3,
  "after_tokens": 52,
  "saving_tokens": 42,
  "covered_units": ["W12", "W13", "W31", "W32"]
}
```

E. Scope Optimization and Workflow Packing

For each normalized rule, a bottom-up dynamic program compares keeping copies in child scopes with placing one copy at an ancestor. A placement is infeasible when a reachable child does not require the rule or contains an unencoded conflict. The DP returns the minimum-cost feasible placement.

Workflow paths are represented by action identifiers that include action type, tool name, required arguments, and guard class. Re-Pair proposes repeated adjacent pairs; prefix/suffix tries identify shared branch segments. A candidate must have at least two non-overlapping occurrences, compatible entry/exit behavior, and positive saving under Eq. (9). Candidate selection is a weighted set-packing problem. The default implementation greedily selects saving per covered token, then applies pairwise exchange. An exact integer program is provided for small synthetic instances to measure the approximation gap.

F. Structural Audit and Conservative Recovery

The audit parser does not see the original skill or expected contract. It independently parses the rendered skill, after which a deterministic diff checks trigger polarity, guards, modality, workflow reachability, tool arguments, and output fields. For each missing element, recovery restores the shortest original source span covering that element and marks it `locked=true`. Future updates may add related content but cannot delete a locked span without explicit user approval.

G. Atomic Online Updates

Zip-on-Write uses a write-ahead log:

1) parse and validate the patch;
2) record proposed operations and savings;
3) apply them to a copy of the sidecar;
4) validate coverage and JSON schema;
5) render a temporary skill;
6) atomically replace state and text; and
7) commit the log entry.

A crash before the final step leaves the previous skill unchanged. Repacking is performed in a separate transaction so failure cannot corrupt patch ingestion.

Appendix C
Contract Schema and Prompts

A. Core Sidecar Schema

```
{
  "interface": {
    "name": "string",
    "purpose": "string",
    "triggers": [{"text":"string","spans":["B1"]}],
    "exclusions": [{"text":"string","spans":["B2"]}]
  },
  "scopes": [{
    "id": "scope-id",
    "parent": "scope-id|null",
    "guard": "string|true",
    "source_blocks": ["B3"]
  }],
  "workflow": [{
    "id": "W1",
    "scope": "scope-id",
    "kind": "action|decision|loop|fallback|stop",
    "action": "string",
    "tool": "string|null",
    "required_args": ["string"],
    "next": ["W2"],
    "spans": ["B4"],
    "confidence": 0.0
  }],
  "rules": [{
    "id": "C1",
    "scope": "scope-id",
    "modality": "must|must_not|prefer",
    "predicate": "string",
    "guard": "string|true",
    "spans": ["B5"],
    "locked": false,
    "confidence": 0.0
  }],
  "output": {
    "type": "string",
    "fields": [{
      "name":"string",
      "required":true,
      "validation":"string",
      "spans":["B6"]
    }],
    "termination": ["string"]
  },
  "evidence": [{
    "kind": "example|rationale|template|background",
    "supports": ["W1", "C1"],
    "adds_unique_unit": false,
    "source_blocks": ["B7"]
  }],
  "shared_procedures": [{
    "id": "P1",
    "actions": ["W1", "W2"],
    "occurrences": [["W1", "W2"], ["W9", "W10"]],
    "saving_tokens": 24
  }],
  "residual": [{
    "source_block": "B8",
    "reason": "AMBIGUOUS|UNIQUE_EVIDENCE|USER_LOCKED"
  }]
}
```

B. One-Shot Extraction Prompt

System Prompt

```
You are a parser for reusable agent skills. Convert the numbered source blocks into the supplied
contract schema. Extract only requirements supported by cited block IDs. Preserve negation, quantifiers,
branch guards, action order, tool names, required arguments, error handling, and output fields. Do
not compress or summarize. An example adds a unique unit only when it is the sole source of a branch,
command, constraint, field, value restriction, or formatting rule. If type or scope is uncertain, place
the source block in residual with reason AMBIGUOUS. Return JSON only.
```

The host rejects unknown source IDs, uncited units, dangling workflow edges, output fields without evidence, and required arguments not present in the cited block.

C. Patch Prompt

System Prompt

```
Parse the proposed skill patch into the same contract schema. For each unit, identify the narrowest
supported scope and cite patch block IDs. Compare only with the supplied existing units. Propose
one operation: ABSORB, REFINE, EXTEND, or REFACTOR. Preserve every conflict as an explicit guarded
exception. Do not edit unrelated scopes. Return JSON only; the host recomputes all savings and decides
whether to apply the operation.
```

D. Independent Audit Prompt

System Prompt

```
Parse this rendered skill into the contract schema. Do not use the original skill or an expected answer.
Preserve exact triggers, exclusions, guards, modalities, tool arguments, workflow edges, required
output fields, and termination conditions. Return JSON only.
```

E. Prior and Post-Compressed Skill Example

The example shows a LiveMath skill after SkillOpt-based self-evolution for qwen3.6-plus; one-shot SkillZip compresses it from **936 to 638 tokens (32% saving) with nearly perfect preservation of performance**. Over self-evolution rounds, the SkillOpt evolver repeatedly re-appended some overlapping guidance, making the skill carry *redundancy*: the "output only the single valid value / discard extraneous roots" rule and the "verify each intermediate product and sum" rule each appear **twice**, and two near-identical "verify each candidate against all constraints" rules coexist. SkillZip *removes these duplicates, folds the two near-paraphrases into one, and normalizes the free-text* `Approach` *into a compact numbered* `Workflow`, while **preserving every distinct output contract verbatim in meaning**—the `\boxed{}` format, set/expression notation, ascending-sort, factorial-magnitude, and extraneous-root-discarding requirements all survive. Because compression *only re-expresses or de-duplicates content already covered by the extracted contract*, the result is **strictly shorter yet performance equivalent**.

The following is the uncompressed skill.

```
## Name
Competition-style math solver.

## Description
Solve competition-style mathematics problems accurately and return the final
answer in the exact required format.

## When to use
- Any single-answer math problem: algebra, number theory, combinatorics,
  geometry, calculus, probability, or discrete math.

## Approach
1. Restate the problem in your own words; list every given quantity and the
   exact quantity to find.
2. Classify the problem and choose a suitable method (direct computation, algebraic manipulation, casework, invariants,
     symmetry, recursion, or a known theorem) and briefly justify the choice.
3. Introduce explicit notation for all unknowns and constraints before computing.
4. Solve step by step, keeping exact values (fractions, radicals, symbolic
   constants such as pi or e) rather than rounding early.
5. Verify the result: substitute back into the original relations, sanity-check
   magnitude and units, and confirm every constraint and edge case holds.

## Rules
- Never round intermediate results unless the problem explicitly asks for a
  decimal approximation.
- Reduce fractions to lowest terms and rationalize denominators where standard.
- For counting problems, decide ordered vs unordered and with vs without
  replacement before summing, and enumerate disjoint cases exhaustively.
```

```
- State any assumption you make when the problem is ambiguous, then proceed with the most standard interpretation.

## Output
- Show concise working, then put the final answer on its own line as \boxed{...}.
- Give the answer in simplest exact form (e.g., 1/2 not 0.5) unless a decimal is explicitly requested.
- For a set answer use set notation \{...}; for an expression, output the
  simplified expression itself rather than a numeric approximation.
- If the problem requests multiple answers, list all distinct values separated by commas inside a single \boxed{} command
     (e.g., \boxed{a, b}).
- If the problem asks for a single specific value (e.g., maximum, minimum, unique solution) or implies a unique answer,
    output ONLY that single valid value. Strictly discard any extraneous candidates, negative roots, or intermediate
    results that do not satisfy all constraints (such as positivity or domain restrictions); do not list them in the
    final \boxed{}.
- For problems requesting multiple answers, strictly verify that each listed value satisfies all original problem
    constraints and definitions; discard any extraneous roots or candidates that fail validation. When listing valid
    multiple answers, strictly sort them in ascending numerical order before placing them in \boxed{}.
- For problems involving factorials, permutations, or large integer products, explicitly compute the magnitude or leave
    in factorial notation; never default to 1 or small integers unless rigorously derived.
- For multi-step arithmetic or large number calculations, explicitly verify each intermediate product and sum to prevent
    off-by-one or digit errors.
- If the problem asks for a single specific value (e.g., maximum, minimum, unique solution) or implies a unique answer,
    output ONLY that single valid value. Strictly discard any extraneous candidates, negative roots, or intermediate
    results that do not satisfy all constraints (such as positivity or domain restrictions); do not list them in the
    final \boxed{}.
- For problems with multiple valid answers, strictly verify each candidate against all problem constraints (e.g.,
    positivity, integer requirements, domain definitions) and discard any that fail validation. If a specific subset of
    solutions is requested (e.g., positive integers), exclude all others.
- For multi-step arithmetic or large number calculations, explicitly verify each intermediate product and sum to prevent
    off-by-one or digit errors.
```

The following is the corresponding compressed skill by SkillZip.

```
## Name
Competition-style math solver.

## Description
Solve competition math accurately; return exact required format.

## When to use
- Any single-answer math problem: algebra, number theory, combinatorics, geometry, calculus, probability, discrete math.

## Rules
- Never round intermediate results unless problem explicitly asks decimal approximation.
- Reduce fractions lowest terms; rationalize denominators where standard.
- For counting problems, decide ordered vs unordered and with vs without replacement before summing; enumerate disjoint
    cases exhaustively.
- State any assumption you make when the problem is ambiguous, then proceed with the most standard interpretation.

## Workflow
1. Restate problem; list every given and exact quantity to find.
2. Classify problem; choose suitable method (direct computation, algebraic manipulation, casework, invariants, symmetry,
    recursion, known theorem); justify.
3. Introduce explicit notation all unknowns and constraints before computing.
4. Solve step by step; keep exact values (fractions, radicals, symbolic constants such as pi or e) not rounding early.
5. Verify result: substitute original relations, sanity-check magnitude and units, confirm every constraint and edge case
     holds.

## Output
- Show concise working; put final answer own line as \boxed{...}.
- Give simplest exact form (e.g., 1/2 not 0.5) unless decimal explicitly requested.
- Set answer use \{...\}; expression output simplified expression itself not numeric approximation.
- If problem requests multiple answers, list all distinct values separated commas inside single \boxed{} command (e.g., \
    boxed{a, b}).
- When problem asks single specific value or implies unique answer, output ONLY that single valid value; strictly discard
     extraneous candidates, negative roots, intermediate results not satisfying all constraints (e.g., positivity, domain
     restrictions); do not list final \boxed{}.
- When problems involve factorials, permutations, large integer products, explicitly compute magnitude or leave factorial
     notation; never default 1 or small integers unless rigorously derived.
- When multi-step arithmetic or large number calculations, explicitly verify each intermediate product and sum.
- Strictly verify each listed value satisfies all original constraints and definitions (e.g., positivity, integer
    requirements, domain definitions); discard extraneous roots or failing candidates; if specific subset requested (e.g
    ., positive integers), exclude others. When listing valid multiple answers, strictly sort ascending numerically
    before placing in \boxed{}.
```

Appendix D
Additional Theory and Analysis

A. Why the Objective Is More Than Semantic Deduplication

Semantic deduplication answers whether two clauses have similar meanings. The shortest-explanation objective asks a different question: after paying for a shared definition, references, scope notation, and exceptions, is the shared representation actually shorter? It can reject a plausible merge when exception encoding is verbose, and it can accept a workflow abstraction whose

occurrences are far apart in the document but share the same guarded action pattern. It also compares alternative abstractions that compete for the same source units.

B. Optimization Decomposition

For a fixed candidate set, the optimizer separates into several subproblems. Equivalent-unit clustering uses union–find after conflict filtering. Rule placement is a tree dynamic program. Workflow selection is weighted set packing because overlapping procedures cannot both replace the same action occurrence. The implementation uses greedy selection with pairwise exchange and provides an exact integer program for benchmark instances. The exact solver measures the approximation gap but is not used in the main efficient configuration.

C. Boundary of the Guarantee

Proposition IV.1 protects the contract produced by the parser; it does not prove that a language model has perfectly interpreted arbitrary natural language. SkillZip makes this limitation visible through cited source blocks, parser confidence, locked residuals, independent structural audit, and per-type parser evaluation. The intended failure mode is to retain too much text, not to silently delete an uncertain rule.

D. Idempotence Under a Stable Parse

For fixed parser output, candidate set, length model, renderer, and deterministic tie breaking, let $\mathsf{Zip}(S)$ be the selected rendered skill. If rendering and reparsing recover the same covered contract, a second pass introduces no new candidate that was absent from the first minimization. Therefore $\mathsf{Zip}(\mathsf{Zip}(S)) = \mathsf{Zip}(S)$. This property will be tested empirically by applying one-shot compression twice and reporting token and contract differences.